Question 1 [9 + 6 Marks]

(A) [9 Marks] Write a non-member function **sumOpposite** that has two parameters **st1** and **st2** of type **stackType**. The stack **st2** is initially empty. The function adds the first inserted element (bottom element) and the last inserted element (top element) of **st1** and pushes the result in **st2**. The function continue the same process by computing the summation of the second inserted element in **st1** and the element inserted before the last in **st1**, and pushing the result in **st2**, and so on.

The function returns false if **st1** is empty, else the function returns true at the end. Assume that **st1** has even number of elements and you need not have to check for that. All the elements of **st1** should be in the original relative order.

Assume that class **stackType** is available for use. Use only common stack operations such as push, pop, top, isEmptyStack, isFullStack, operator= and copy constructor.

Function prototype:

```
bool  sumOppsite(stackType<Type>& st1, stackType<Type>& st2);
```

Example:

Stack st1 : 1 3 4 12 8 6 2 5 (top)

Stack st2 after function call : 20 10 5 6 (top)

Note that 1 + 5 = 6, so first push 6 in st2.

Next, 3 + 2 = 5, so push 5 in st2.

Next 4 + 6 = 10, so push 10 in st2.

Finally, 12 + 8 = 20, so push 20 in st2.

```
template <class Type>
bool  sumOppsite( stackType<Type> & st1, stackType<Type> & st2)
{
  if (st1.isEmptyStack()) return false;
  stackType<Type> st3, temp(st1);
  while( ! temp.isEmptyStack())
  {
     st3.push(temp.top());
     temp.pop();
  }
  temp = st2;
  Type R;
  while( ! temp.isEmptyStack())
  {
    R = temp.top() + st3.top();
    st2.push(R);
    st3.pop();
    temp.pop();
  }
  return true;
}
```

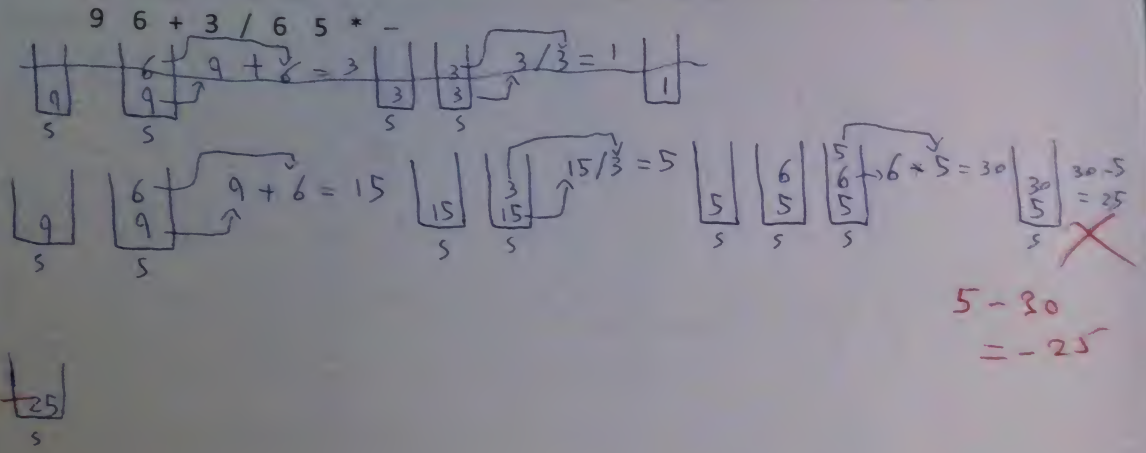
You should have half way

7½

5

(B) [6 Marks] Consider the following <u>postfix expression</u>. Use stack to evaluate it and show all the push and pop operations by clearly drawing the stack status.

9 6 + 3 / 6 5 * −



5 - 30
= - 25

Question 2 [10 Marks]

Write a non-member function called **negativeFirst** that receives a queue object **Qu** of type **queueType** as parameter. The function takes all negative elements found in **Qu** and places them at the front of the queue, the order of other elements in the queue will remain unchanged.

Example:

Qu before function call:

Qu: 2 -5 7 -12 -20 22 4

Qu after function call:

Qu: -5 -12 -20 2 7 22 4

10

Function prototype:

```
void  negativeFirst(queueType<Type>&  Qu);
```

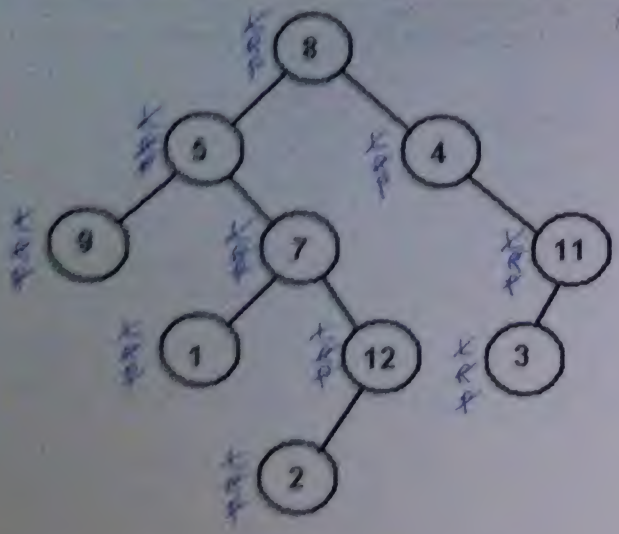
You may use common queue operations such as addQueue, deleteQueue, front, back, isEmptyQueue, isFullQueue, operator= and copy constructor in your function.

```
template <class Type>
void negativeFirst(queueType<Type>& Qu)
{
  queueType<Type> neg, pos;
  while( Qu
  if ( Qu.isEmptyQueue())
    cout<<"Empty Queue"<<endl;
  else{
   while( !Qu.isEmptyQueue())
   {
     if ( Qu.front() >=0)
     {
       pos.addQueue( Qu.front());
       Qu.deleteQueue();
     }
     else{
         neg.addQueue(Qu.front());
         Qu.deleteQueue();
         }
   }
  while(!neg.isEmptyQueue())
  {
    Qu.addQueue(neg.front());
    neg.deleteQueue();
  }
  while( !pos.isEmptyQueue())
  {
    Qu.addQueue(pos.front());
    pos.deleteQueue();
  }
}
}
```

4

Question 3 [8 + 7 Marks]

(A) For the binary tree given below, answer the following questions:



i. **[2 Marks]** What is the **height** of this binary tree?

5 — 1

ii. **[2 Marks]** List all the leaf nodes of this binary tree.

9, 1, 2, 3 — 2

iii. **[5 Marks]** List the sequence of nodes, if the binary tree is traversed using **post-order traversal**

~~1 2 2 12 7~~

9, 1, 2, 12, 7, 5, 3, 11, 4, 8 — 5

(B) [7 Marks] Write a recursive private member function called `countLeaves` to be included in class `binaryTreeType`. The function counts the number of leaf nodes in the binary tree and returns this count. This function is called from a public member function `treeCountLeaves`, given as follows:

```
template<class Type>
int  binaryTreeType<Type>::treeCountLeaves( )
{
     return    countLeaves( root );
}
```

Function prototype:

```
int  countLeaves(nodeType<Type>  *p);
```

```
template <class Type>
int binaryTreeType<Type>::countLeaves ( nodeType<Type> *p)
{
  if ( p == NULL)
    return 0;
  else if ( p->lLink == NULL && p->rLink == NULL)
      return 1;
  else
    return 0 + countLeaves(p->lLink) + countLeaves(p->rLink);
}
```